Chapter 1

Object Oriented Programming

1.1 Object oriented approach

Definition 1.1.1: Object Oriented Programming

Object oriented programming (OOP) is a imperative programming programming paradigm where instructions are grouped together with the part of the state they operate on.

The main **structural components** of all systems are

- Objects: Object is something that takes up space in the real or conceptual world with which somebody may do things. An object is an instance of a class. An object has three main characteristics:
  - Name (or ID): Unique identifier for the object.
  - State: Attributes that describe the object.
  - Operations (or behavior): Actions that the object can perform.
- Class objects: Class is the blueprint of an object.

Main **characteristics** of the approach are

- Encapsulation: The mechanism of hiding the implementation of the object. Only the necessary details are exposed to the user.
- Abstraction: A principle which consists of ignoring the aspects of a subject that is not relevant for the present purpose. It is the process of simplifying complex systems by modeling classes based on the essential properties and behaviors an object must have.
- Inheritance: Mechanism by which one class can inherit attributes and methods from another class. Making new classes from existing ones.
- Polymorphism: Ability to present the same interface for different data types. Same function name but different signatures being used for different types.

1.2 Class

Definition 1.2.1: Class Diagram

A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

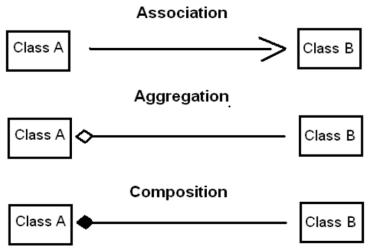
Elements of a **class**:

- Class Name (ID Class): The name of the class. Nouns associated with the textual description of a problem. Singular.

- Attributes: The properties or characteristics of the class. Types include Real, Integer, Text, Boolean, Enumerated, etc.
- Operations (Methods): The functions or operations that can be performed on the class. Behaviors of the class.

Relations between classes:

- Association: A relationship between two classes that indicates how objects of one class are connected to objects of another class.
- Aggregation: A special type of association that represents a "whole-part" relationship between classes.
- Composition: A stronger form of aggregation that implies ownership and a whole-part relationship where the part cannot exist independently of the whole.
- Generalization: A relationship between a more general class (superclass) and a more specific class (subclass) that indicates inheritance.



1.2.1 Class and Method in Python

```python
class Person:
    pass # An empty class definition

p = Person()
print(p)  # Output: <__main__.Person object at 0x...>
```

Listing 1.1: Empty class definition in Python

```python
"""Define class with method"""
class Person:
    def speak(self):
        print("Hello!")

"""Create object and call method"""
p = Person()
p.speak()  # Output: Hello!
```

Listing 1.2: Method

```python
"""
The __init__ method is a special method in Python classes.
It is a method that Python calls when you create a new instance of this class.
"""
class Person:
    def __init__(self, name):
        self.name = name  # Initialize the name attribute
    def speak(self):
        print('Hello, my name is', self.name)

p = Person('Carlos')
p.speak()  # Output: Hello, my name is Carlos
```

Listing 1.3: Attributes and __init__ method

Note:

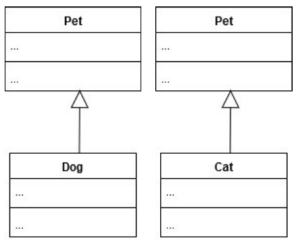
The first argument of every class method, including init, is always a reference to the current instance of the class. By convention, this argument is always named self.

```
class Pet(object):
    def __init__(self, name, species):
        self.name = name
        self.species = species
    def getName(self):
        return self.name
    def getSpecies(self):
        return self.species
    def __str__(self):
        return "%s is a %s" % (self.name, self.species)
```

Listing 1.4: Class Pet with attributes and methods

```
class Dog(Pet):

"""Dog inherits from Pet"""

    def __init__(self, name, chases_cats):
        # Call the parent class (Pet) constructor
        Pet.__init__(self, name, "Dog")
        self.chases_cats = chases_cats

    def chasesCats(self):
        return self.chases_cats

class Cat(Pet):

"""Cat inherits from Pet"""

    def __init__(self, name, hates_dogs):
        # Call the parent class (Pet) constructor
        Pet.__init__(self, name, "Cat")
        self.hates_dogs = hates_dogs

    def hatesDogs(self):
        return self.hates_dogs
```

Listing 1.5: Inheritance



Example 1.2.1 (Inheritance)

myPet = Pet("Boby", "Dog")
myDog = Dog("Rex", True)

- isinstance(myDog, Pet) # Returns: True
- isinstance(myDog, Dog) # Returns: True

- isinstance(myPet, Pet) # Returns: True
- isinstance(myPet, Dog) # Returns: False

1.2.2 Access Modifiers

There are three types of access modifiers in Python:

- Public: Members (attributes and methods) declared as public are accessible from anywhere.
- Protected: Members declared as protected are accessible within the class and its subclasses. In Python, this is indicated by a single underscore prefix (e.g., _attribute).
- Private: Members declared as private are accessible only within the class itself. In Python, this is indicated by a double underscore prefix (e.g., __attribute).

```
class Person:
    def __init__(self, name, age):
        self.__name = name      # Private attribute
        self.__age = age        # Private attribute

p = Person("David", 23)
p.__name  # This will raise an AttributeError
```

Listing 1.6: Private Attributes

```
class Person:
    def __init__(self, name, age):
        self.name = name      # Public attribute
        self.age = age        # Public attribute

p = Person("David", 23)
p.name  # This will work fine and return "David"
```

Listing 1.7: Public Attributes

```
class Person:
    def __init__(self, name, age):
        self._name = name      # Protected attribute
        self._age = age        # Protected attribute

p = Person("David", 23)
p._name  # This will work, but it's discouraged to access protected members directly
```

Listing 1.8: Protected Attributes

```
class Person:
    def __init__(self, money=0, energy=100):
        self.money = money
        self.energy = energy

    def work(self, hours):
        if self.energy >= hours * 10:
            self.money += hours * 10 # Assume earning $10 per hour of work
            self.energy -= hours * 10
            print (f"Worked for {hours} hours. Money increased to ${self.money}, energy
                decreased to {self.energy}.")
        else:
            print("Not enough energy to work.")
```

Listing 1.9: Example Usage

Chapter 2

Data Analysis and Statistics

2.1 Numpy

Definition 2.1.1: Numpy

Numpy is a fundamental package for scientific computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures efficiently.

2.1.1 Shape, rank, and size

```
    c = np.array([1, 2, 3, 4])  # Create a 1D array
    print(type(c))  # Output: <class 'numpy.ndarray'>

    b = np.array([[1, 2, 3], [4, 5, 6]])  # Create a 2D array
    shape = b.shape  # Output: 2, 3
    rank = np.ndim(b)  # Output: 2
    size = b.size  # Output: 6
```

Listing 2.1: Shape, rank, and size

2.1.2 Maniputating arrays

Reshaping arrays means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping, we can add or remove dimensions or change number of elements in each dimension.

```
    arr = np.array([4, 4, 3, 4, 5, 6, 7, 9, 0, 10, 17, 12])
    newarr = arr.reshape(3, 4)  # Reshape to 3 rows and 4 columns
    print(newarr)
    # Output:
    # [[ 4  4  3  4]
    #  [ 5  6  7  9]
    #  [ 0 10 17 12]]
```

Listing 2.2: Reshaping an array

We can also access an array element and change value to an array.

```
    a = np.array([10, 20, 30, 40, 50])
    a[2] = 50  # Change the third element to 50
    print(a)  # Output: [10 20 50 40 50]
```

Listing 2.3: Accessing and changing array elements

```
a = np.zeros((2,2))  # Create a 2x2 array of zeros
b = np.ones((3,3))   # Create a 3x3 array of ones
c = np.full((2,3), 7)  # Create a 2x3 array filled with 7s
d = np.eye(3)    # Create a 3x3 identity matrix
```

Listing 2.4: Creating special arrays

Joining NumPy arrays means putting contents of two or more arrays into a single array.

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = np.concatenate((a, b))  # Join a and b
print(c)  # Output: [1 2 3 4 5 6]

d = np.stack((a, b), axis=0)  # Stack a and b vertically
print(d)
# Output:
# [[1 2 3]
#  [4 5 6]]

e = np.stack((a, b), axis=1)  # Stack a and b horizontally
print(e)
# Output:
# [[1 4]
#  [2 5]
#  [3 6]]
```

Listing 2.5: Joining NumPy arrays

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
# Output:
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]

b = a[:3, 1:3]  # Slicing rows 0-2 and columns 1-2
print(b)
# Output:
# [[ 2  3]
#  [ 6  7]
#  [10 11]]
```

Listing 2.6: Slicing NumPy arrays

Problem 2.1: Slicing arrays

b[0, 0] = 20
What happens to array a?
Answer: Array a is also changed because b is a view of a.

Problem 2.2: Slicing arrays

How to get a copy of a slice of an array so that changes to the slice do not affect the original array?
Answer: Use the .copy() method to create a copy of the slice.

```
b = a[:3, 1:3].copy()  # Create a copy of the slice
b[0, 0] = 20  # Change the first element of b
```

```
print(a)  # Array a remains unchanged
# Output:
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
```

Listing 2.7: Creating a copy of a slice

2.1.3 Handling indexes

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print(a)
# Output:
# [[ 1  2  3]
#  [ 4  5  6]
#  [ 7  8  9]
#  [10 11 12]]

b = np.array([0, 2, 0, 1])
print(a[np.arange(4), b])  # Access elements using array of indices
# Output: [ 1  6  7 11]

a[np.arange(4), b] += 10  # Increment selected elements by 10
print(a)
# Output:
# [[11  2  3]
#  [ 4  5 16]
#  [17  8  9]
#  [10 21 12]]
```

Listing 2.8: Fancy indexing

```
b = 1
b += 10
print(b)  # Output: 11

b = 1
b = b+10
print(b)  # Output: 11
```

Listing 2.9: Boolean indexing

```
grades = np.array([14, 12, 13, 14, 15, 14, 14])
x = np.where(grades == 14)  # Find indices where grades are 14
print(x)  # Output: (array([0, 3, 5, 6]),)
```

Listing 2.10: Searching arrays

2.1.4 Random

Definition 2.1.2: Random

Random number does not mean a different number each time. Random means something that cannot be predicted logically.

```
from numpy import random

# Generating random numbers within an array
e = np.random.random((4, 4)) # Create a 4x4 array of random floats in [0.0, 1.0)
```

```
    # Generating random numbers
    x = random.randint(100)  # Generate a random integer from 0 to 99

    # Generating arrays of random numbers
    x = random.randint(100, size=(5)) # Generate an array of 5 random integers from 0 to 99
    x = random.randint(100, size=(3, 4)) # Generate a 3x4 array of random integers from 0
        to 99
    x = random.rand(5)  # Generate an array of 5 random floats in [0.0, 1.0)
```

Listing 2.11: Generating random numbers

Definition 2.1.3: Random data distribution

A random distribution is a set of random numbers that follow a certain probability density function (PDF).

- Normal distribution (random.normal)
  - loc: (mean) where the peak of the bell curve is located
  - scale: (standard deviation) how flat and wide the bell curve is
  - size: output shape
- Binomial distribution (random.binomial)
  - n: number of trials
  - p: probability of success on each trial
  - size: output shape

```
    # Normal Distribution
    x = random.normal(loc=0.0, scale=1.0, size=(2,3))  # Generate a 2x3 array of random
        numbers from a normal distribution
    print(x)
    # Output:
    # [[ 0.49671415 -0.1382643   0.64768854]
    #  [ 1.52302986 -0.23415337 -0.23413696]]

    # Binomial Distribution
    x = random.binomial(n=10, p=0.5, size=(3,4))  # Generate a 3x4 array of random numbers
        from a binomial distribution
    print(x)
    # Output:
    # [[5 6 4 3]
    #  [4 7 6 4]
    #  [6 3 5 4]]
```

Listing 2.12: Generating random numbers from different distributions

2.2 Pandas

Definition 2.2.1: DataFrame

Labelled data structure with columns of potentially different types. Like a spreadsheet or SQL table, or a dict of Series objects.

```
    d = {
        'col1': [1,2,1,3,1,2],
        'col2': [1,2,3,4,5,6]
        }
    df = pd.DataFrame(data=d)

    df.count() # Count non-NA cells for each column
    df['col1'].value_counts() # Count unique values in col1
    df['col1'][1] = 5 # Change value in col1 at index 1 to 5
```

Listing 2.13: Creating and manipulating a DataFrame

Problem 2.3: Copy DataFrames

col1 = df['col1'] # Accessing a single column
col1[2] = 99
What is the result in col1 and df?
Answer: Both col1 and df are changed because col1 is a view of df.

Problem 2.4: Copy DataFrames

new_col1 = col1.copy()
new_col[2] = 9999

What is the result in new_col1 and df?
Answer: Only new_col1 is changed, df remains unchanged because new_col1 is a copy of col1.

```
    df = pd.read_csv('data.csv')  # Read data from a CSV file into a DataFrame
    df.to_csv('output.csv')  # Write DataFrame to a CSV file
```

Listing 2.14: Read and save

```
    df.head()  # Display the first 5 rows of the DataFrames
    df.info()  # Display a summary of the DataFrame
    df.describe()  # Generate descriptive statistics
    df.columns  # List all column names
```

Listing 2.15: DataFrame information

```
    df.at(2, 'col1')  # Access a single value for a row/column label pair
    df.iloc[2, 0]  # Access a single value for a row/column pair by integer position
    df.xs(2)  # Returns cross-section (row) at index 2
    df.loc[:, 'col1']  # Access a group of rows and columns by labels or a boolean array
```

Listing 2.16: Accessing DataFrame elements

Example 2.2.1 (Access to row and columns)

- Cells: df.iloc[195][0]
- Rows: df.iloc[[195], :]
- Columns: df.loc[:, 'col1']

```
    df1=df2 # Assignment
    copydf = df.copy(deep=False) # Shallow copy
    copydf = df.copy(deep=True)  # Deep copy
```

Listing 2.17: Copying DataFrames

```
    # Convert column to numeric, setting errors to NaN
    df['GDP'] = pd.to_numeric(df['GDP'], errors='coerce')

    # Suppose you have , instead of . in the numbers
    df['GDP'] = df['GDP'].str.replace(',', '.')

    # Suppose you have $ or € symbols in the numbers
    df['GDP'] = df['GDP'].str.replace('$', '', regex=True).str.replace('€', '', regex=True)

    # Drop rows with any NaN values
    df.dropna(inplace=True)
```

Listing 2.18: Data cleaning and conversion

```
    X.mean()  # Calculate mean of each column
    X.median()  # Calculate median of each column
    X.mode()  # Calculate mode of each column
    X.std()  # Calculate standard deviation of each column
    X.var()  # Calculate variance of each column
    X.corr()  # Calculate correlation matrix
    X.cov()  # Calculate covariance matrix
    X.max()  # Calculate maximum of each column
    X.min()  # Calculate minimum of each column
    X.kurt()  # Calculate kurtosis of each column
    X.skew()  # Calculate skewness of each column
```

Listing 2.19: Statistical methods